

The Ambiguity Criterion in Software Design

Álvaro García

E-mail: alvarillogp@gmail.com

Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software

Facultad de Informática (Universidad Politécnica de Madrid)

Campus de Montegancedo, 28660 Boadilla del Monte, Madrid (SPAIN)

Tel: (+34) 913366926

Nelson Medinilla

E-mail: nelson@fi.upm.es

ABSTRACT

In this paper we introduce the *ambiguity criterion* in software design. There is neither an objective criterion to measure the quality of a software design nor a general strategy to manage complexity and uncertainty. Ambiguity, as a specific kind of uncertainty, is a powerful tool to face uncertainties in the environment as well as a descriptive complexity reduction mechanism. Ambiguity is present at every level and stage of the software process, which has never been acknowledged by the software engineering community. We show that many of the sophistications in software design, from the use of variables in assembly language to the most recent patterns in object-oriented design, are in fact uses of ambiguous relationships among software elements. Ambiguity can help to establish the basis of a software design theory.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques – *modules and interfaces, object-oriented design methods.*

D.2.3 [Software Engineering]: Coding Tools and Techniques – *object-oriented programming, structured programming, top-down programming.*

D.3.3 [Programming Languages]: Language Constructs and Features – *abstract data types, classes and objects, frameworks, inheritance, modules and packages, patterns, polymorphism, procedures, functions and subroutines.*

General Terms

Design, Languages, Theory.

Keywords

Ambiguity, Indifference, Complexity, Uncertainty, Programming Paradigms.

1. INTRODUCTION

Ambiguity has always been considered as a harmful aspect we need to eradicate when trying to develop software. Often

connected to non-deterministic behavior, it is contrary to the definition of *algorithm* [1], and other definitions of computability. The term *unambiguous* in this context refers to a mathematical formalism which sustains computability theory. Nevertheless, ambiguous representations of deterministic systems arise spontaneously [2] and are consistent with computability theory. But the idea of eradicating ambiguity has been erroneously extended to software engineering. It is clearly stated in [3] that any requirement in the Software Requirements Specification (SRS) must be unambiguous. There should be no problem with representing deterministic and computable systems with some degree of ambiguity, if they are truly computable.

For example, when using a variable in our code, we are assuming the specific value is uncertain. The benefits of variables appear as soon as the specific value is not known. Trying to determine which value is needed in every moment would be counterproductive. Although writing that value in the code would be consistent, and in few cases can increase computational efficiency, it forces us to renounce certain sophisticated mechanisms, such as conditional statements and loops, and descriptive complexity will explode, making our program an obfuscated *spaghetti code*.

In that situation we will be forced to invent some solutions that generalize the program, in order to be suitable for a variety of execution cases. Although it is possible, it would be a comeback to a less sophisticated programming paradigm, similar to that of the *Turing Machine* (TM), limiting our capacity to face large and complex systems. The mechanism invented would be a recurrent and general solution that reduces descriptive complexity on an ambiguity basis. The variables are one of those mechanisms, provided as native in almost every high-level programming language.

In this paper we use the indifference to define ambiguity in software design. We introduce the *ambiguity criterion*, which determines that ambiguity reduces descriptive complexity and faces the uncertainties. We also enumerate three strategies that introduce ambiguity in the designs: *indirection*, *type abstraction*, and *late delegation*. Finally, we revise software patterns with reference to ambiguity, adding a new perspective to the usual definitions they already have.

2. INDIFFERENCE AND THE AMBIGUITY CRITERION

We define ambiguous relationships as follows: “let A be a module that depends on a module B. If we substitute B for B' without altering the correctness of A then we say that A has an ambiguous (or indifferent) relationship to B (and also to B')” (Figure 1).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Conference '07, November 5–9, 2007, Atlanta, Georgia, USA.
Copyright 2007 ACM 1-58113-000-0/00/0004...\$5.00.

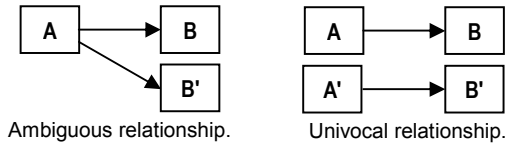


Figure 1. Ambiguous and univocal relationships.

Ambiguous relationship means indifferent relationship. *Liskov's substitution principle* [4] is a particular instance of indifference, since typing and subtyping do not take part in it and there is no restriction about what a module is. It could be any set of software or hardware elements identified by a noun that acts like a unity in some level of abstraction.

We can, in a reverse way, define a univocal relationship as follows: "let A be a module that depends on a module B. If we substitute B for B' and we need to alter A, obtaining A', to preserve the correctness then we say that A has a univocal relationship to B" (Figure 1).

The ambiguous relationship lets a module to vary in some range without altering the modules depends on it. This may be one of the key concepts to achieve flexible and maintainable systems [5]. Indifference reduces descriptive complexity, since every alternative does not multiply the number of modules pointing to these alternatives (Figure 1).

The *ambiguity criterion* states that ambiguity is desirable because it can reduce the descriptive complexity. It also results in less development and maintenance effort and makes the system more flexible. Ambiguity allows our design to be alternative impervious. Changing from one alternative to other has near a zero-cost.

3. COMPLEXITY AND AMBIGUITY

In a software system, we can identify two kinds of complexity: uncertainty complexity and descriptive complexity [6]. Uncertainty complexity is the amount of information needed to resolve any uncertainty about the system. Descriptive complexity is the amount of information needed to describe the system [7].

Ambiguity is a kind of uncertainty. It refers to the presence of alternatives, or in other words, relationship from one to many [7]. But ambiguity has connotations of being imprecise, fuzzy or vague, in contrast to exactitude and completeness. Therefore ambiguity has been avoided in many development methodologies. We define it as indifference, indeterminacy or indistinctiveness. These terms do not have such negative connotations, and can be suitable to prevent misunderstanding and prejudice.

When considering uncertainty complexity, solutions containing uncertainty become apparent. These solutions may have some ambiguity, which can reduce descriptive complexity. The introduction of the uncertainty axe enriches the software universe, showing some powerful new solutions which were concealed in the linear universe of descriptive complexity [6].

4. ABSTRACTION AND TOP-DOWN REASONING

The Merriam-Webster dictionary defines *abstract* as follows [8]:

"**abstract**: to consider apart from application to or association with a particular instance."

Abstraction considers the essence of something while omitting the details of the instances of that thing. By means of abstraction we establish an ambiguity from the essence to the instances. The essence represents many instances. It is obviously a particular case of indifference (or indeterminacy or indistinctiveness).

In the software development universe it is common to identify abstraction with modularization and *top-down* reasoning, which is a kind of *divide and conquer* strategy [9]. Some people think they can find the abstractions above and descend to the instances following that strategy, but as we see next this is an erroneous intuition. *Bottom-up* reasoning often yields to the same design, except that instances are supposed to come first and abstractions later.

In both cases the axe considered is *extension*. In top-down reasoning we face first the bigger elements, and then the smaller ones. In bottom-up reasoning is the other way round. The operations considered are *aggregation* and *segregation*. They result in a tree. We need a breadth-first traversal to not leave out any detail that any lower-level element could depend on. That leads us into an exhaustive walk, making it harder to move quickly from one level to the lower ones.

With abstraction we focus on the axe of *generality*. Regardless of the size of the elements considered, we propose entities that can represent many of them. The relationships between entities of that kind do not adjust to a tree, but rather to a general graph. It is easier to do a deep-first traversal of that graph, because in many cases those relationships do not involve specific elements (instances) but abstractions (essences). It is possible to travel deep into the instances without affecting the other essences. The operations considered are *abstraction* and *concretization*. *Aggregation* and *segregation* are also considered, but as complexity innocuous as possible, thanks to indifference.

In Figure 2 we can see an enterprise model using top-down reasoning. There is no generality, as elements only differ in size. Nothing prevents the relationships to be univocal, because every element is an instance.

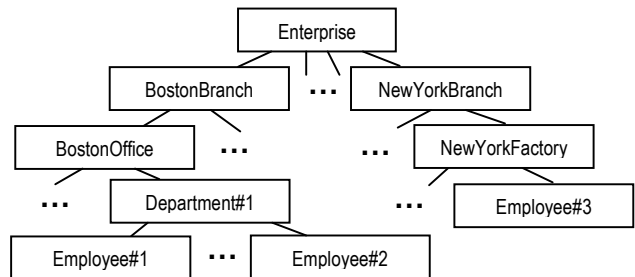


Figure 2. Enterprise model using top-down reasoning.

Aggregation yields to one-to-many relationships without indifference (without ambiguity). Thus, there is no reduction of descriptive complexity. It is only distributed among the different subsystems. Descriptive complexity tends to grow linearly with system size and exponentially with the detail level considered.

In Figure 3 we can see an enterprise model using abstraction instead of top-down reasoning. We have a graph, with some tree-like hierarchies. Dependencies tend to occur among the abstractions, but not among instances.

Abstraction yields to one-to-many relationships with indifference (with ambiguity). Thus, combined with abstraction, aggregation is

turned to homogeneous one-to-many relationship, like with Team and Employee in Figure 3. Descriptive complexity tends to be constant when faced with system size and to increase linearly with the detail level considered.

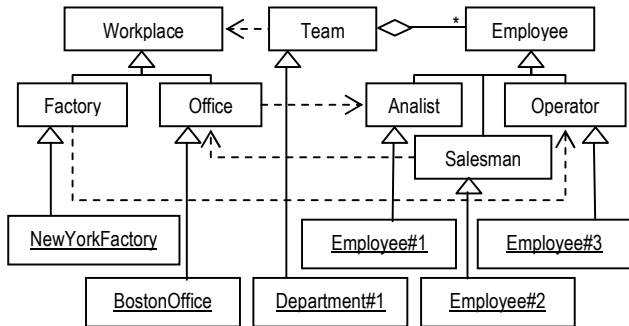


Figure 3. Enterprise model using abstraction.

The figures above do not represent a software system. They are only an example of a real system, with elements and relationships among them. In the case of a software system, the two axes considered (extension and generality) would refer to functionality, code, or whatever software entities we consider. Nevertheless, we can apply the same conclusions no matter what (physical objects or software) we are working with. We do not mean that the software system should reflect reality, but rather the opposite. Copying reality does not help to achieve easy-to-modify-and-maintain systems [5].

5. SOFTWARE ELEMENTS

We are trying to establish ambiguous relationships among modules, and more specifically among software elements. We should enumerate and revise them focusing on ambiguity.

When thinking of a program a lot of entities come to our mind: memory positions, instructions, data, routines, sentences, variables, procedures and functions, pointers, data structures, abstract data types (ADT), methods, members, objects, classes, interfaces, templates, agents, etc.

Almost every sophisticated element incorporates other less sophisticated ones, perhaps with a different name. Thus, object-oriented paradigm integrates variables, functions and data structures, but the new names are members, methods and classes.

We can also enumerate some of the mechanisms present in programming: addressing modes, memory maps for code and data, use of variables, loops, procedure calls, parameter passing, environmental variables, configuration files, definitions, macros, use of pointers, static link libraries, dynamic link libraries, strong typing, weak typing, threads, synchronous event reception, asynchronous event reception, exceptions, polymorphism, late binding, patterns, protocols to communicate agents, etc.

Every programming paradigm offers its own native elements and mechanisms. It is possible and habitual to emulate one sophisticated element or mechanism using less sophisticated ones. Some examples are: using stack frames to implement function calls in assembly language; using the Win32 graphical Application Programming Interface (API), which uses Object-Oriented Programming (OOP) techniques, for the structured C language; emulating some Artificial Intelligence (AI) representations with a general-purpose programming language.

We work with the hypothesis that a sophisticated mechanism involves ambiguous relationships among its components, whether or not it is visible to the programmer. Frequently, elements that are external to our system, like hardware devices or Operating System (OS) components, are involved. However, the origin of the ambiguous relationship is always a software element in our system.

In the next section we will enumerate three basic strategies to introduce ambiguity, and present some examples of well-known sophisticated mechanisms that use them. We will prove that its strength is due to ambiguity, thus reinforcing the *ambiguity criterion*.

6. STRATEGIES FOR AMBIGUITY

If we examine the similarities among the mechanisms exposed above we find at least three general strategies:

- *Indirection*: when the address of the required resource is stored, instead of the resource itself. It is needed to resolve indirection when accessing to the resource. The system is aware of some resource identity, but is indifferent to the actual value of that resource.
- *Type abstraction*: when working with a hierarchy of subtypes, some of them can be substituted for others without affecting to the correct functionality of the system. The system is aware of the supertype of some resource, but is indifferent to the specific subtype of that resource.
- *Late delegation*: when some decision is delayed until execution time, pointing out the module which should make that decision. The system is aware of the strategy used to perform some action, but is indifferent to the time when the action is performed.

6.1 Indirection

Mechanisms that use indirection are: variables instead of *hardcoded values*; pointers instead of variables; definitions, macros, configuration files, environmental variables, header files, procedures, functions, data structures, ADT, objects and libraries instead of *spaghetti code*; Bridge pattern [10]; etc.

We can compare the use of hardcoded values with the use of variables.

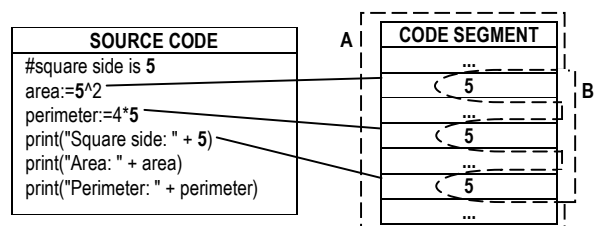


Figure 4. Use of hardcoded values.

With hardcoded values (Figure 4) the value is an operand of an instruction. This value appears scattered in the code. If we need to modify that value, we have to revise the entire code. There is a univocal relationship between the code (A) and the value (B). B is contained in A, so changing B means changing A. The descriptive complexity of changing from one alternative to other grows linearly with the number of times that the value is needed.

With variables (Figure 5) we store the address of the value, not the value itself. Thus, if the value changes the address remains constant. There is an ambiguous relationship between the code (A) and the value (B), so B can vary. The descriptive complexity to change among alternatives remains constant when faced with the number of times that the value is needed (we only need to initialize it once).

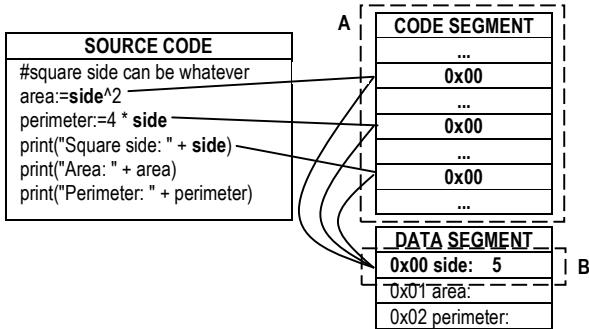


Figure 5. Use of variables.

Using a *Central Process Unit* (CPU) with indirect addressing mode and separate maps for code and data is necessary. It is also the time spent in resolving the indirection. Nevertheless the cost incurred worth the benefits achieved.

6.2 Type abstraction

Mechanism that use type abstraction are: loops instead of *spaghetti code*; procedures, functions and data structures instead of *ad hoc* code writing; weak typing instead of strong typing; ADT instead of *ad hoc* type solutions; interface separation instead of *ad hoc* class implementations; polymorphism instead of multiple methods writing; late binding instead of static binding; Strategy pattern [10]; etc.

We can compare using *ad hoc* class implementation with interface separation:

method by which each of them is accessed. Every new implementation result in a new main program. Descriptive complexity grows linearly with the number of implementations.

With interface separation (Figure 7) we must write the main program according to the interface defined, but we are unaware of what specific subtype is being used. There is an ambiguous relationship from the main program to the subtypes of the interface, so we just need to write it once. Descriptive complexity remains constant when faced with the number of subtypes.

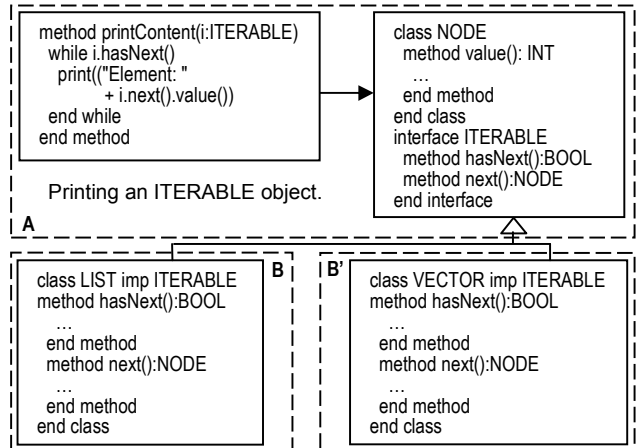


Figure 7. Interface separation.

Interfaces, type hierarchies and late binding have an execution time overcharge, however that cost worth the benefits achieved.

6.3 Late delegation

Mechanisms that use late delegation are: dynamic link libraries instead of static ones; asynchronous event reception instead of synchronous reception; exceptions instead of returning error values; Proxy and Observer pattern [10]; agent communication protocols; etc.

We can compare the use of synchronous event reception with the asynchronous one.

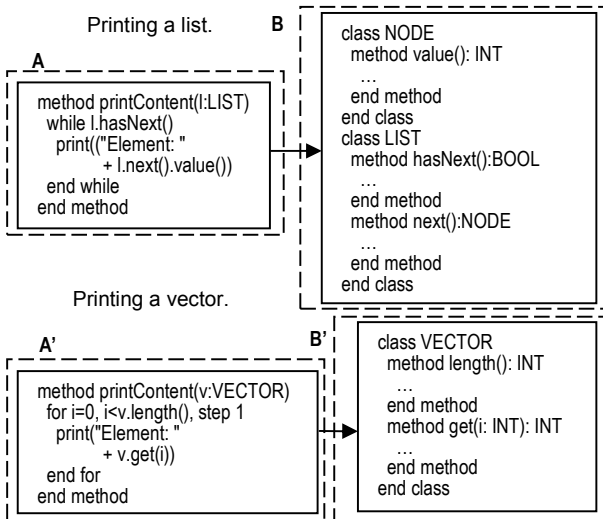


Figure 6. Ad hoc class implementation.

With *ad hoc* class implementations (Figure 6) there are dependencies on the specific classes we use. When printing the contents of a list or vector, the main program must be aware of the

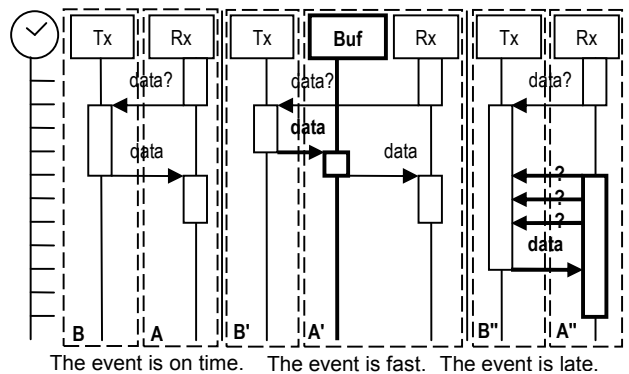


Figure 8. Synchronous event reception.

With synchronous reception we have a *receiver* that waits for an event from a *transmitter* in a fixed amount of time (Figure 8). There is a univocal relationship between the receiver (A) and the transmitter (B). If the event arrives before expected a *buffer* is needed. If it arrives later an occupied waiting scheme is needed. Those solutions are *ad hoc*, only solve the problem partially and

require specific tuning. A scheme with a buffer and occupied waiting in the receiver can be implemented, so the system would be impervious to the reception time. However, one buffer per process waiting should exist, and there is also the occupied waiting waste. The descriptive complexity would grow in proportion to the number of processes waiting and the number of transmitters, so it can not be used as multitask general solution.

With asynchronous reception (Figure 9) we have a blocking and non-occupied waiting scheme. There is an OS component (*planner*) that locks and unlocks the processes and a hardware device (*Interrupt Controller* or IC) that interrupts the CPU when a signal is received. The receiver points to the planner whenever a data is needed. The planner makes the decisions in execution time.

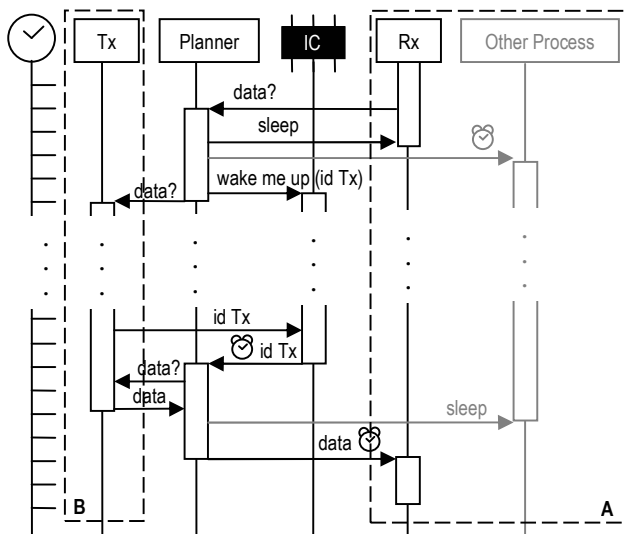


Figure 9. Asynchronous event reception.

Regardless of what time the event is received, the system behaves in a correct manner. There is an ambiguous relationship between the receiver and other processes (A) and the transmitter (B). B can vary in the reception time range without altering A. The IC does occupied waiting, but there is only one in the system. The descriptive complexity remains constant when faced with the number of tasks in the system and the number of transmitters, so it can be used as multitask general solution.

Asynchronous event reception has the overcharge of the planner (included with the OS) and the IC. But like in the above mentioned situations that cost worth the benefits achieved.

7. SOFTWARE PATTERNS

As we have seen in the previous sections, some mechanisms reduce descriptive complexity by establishing ambiguous relationships among modules. To achieve the benefits of indifference, at least the origin of the relationship needs to be a software module. Otherwise we were not talking about programming and software engineering. The possibility of establishing ambiguous relationships to elements other than software extends the power of the mechanisms.

Software patterns are in many cases ambiguity mechanisms. Defining them as recurrent and general solutions that reduce descriptive complexity in an ambiguity basis would be possible.

This definition could be a more concise and intensive than other common definitions.

8. CONCLUSION

In software design, considering the uncertainty complexity together with descriptive complexity shows powerful solutions concealed in the linear universe of descriptive complexity.

Ambiguity, as a kind of uncertainty, is defined in terms of indifference. Ambiguity yields to one-to-many relationships with indifference from the origin to the destinations.

Designs containing ambiguity are consistent with computability theory. A lot of programming mechanisms and software elements make use of it, although not acknowledged by the Software Engineering Community.

Aggregation distributes descriptive complexity among subsystems, ambiguity can reduce it. Ambiguity is a powerful descriptive complexity reduction mechanism that can be combined with aggregation.

Indirection, type abstraction and late delegation are three strategies that introduce ambiguity in software design.

General solutions containing ambiguity arise recurrently in software design, since they reduce descriptive complexity as well as development and maintenance effort. Ambiguity could be a key word in the software pattern definitions.

Ambiguity could help to establish a software design theory.

9. ACKNOWLEDGEMENTS

Thanks to Laura and Liam for their style correction. Thanks to Eugenia for her advice and support.

10. REFERENCES

- [1] Schneider, M. and Gersting, J. *An Invitation to Computer Science*, West Publishing Company, New York, 1995, 9.
- [2] Martin, J. C. *Introduction to Languages and the Theory of Computation*, 2nd edition, McGraw-Hill, 1997, 277–281.
- [3] The IEEE Std 830, *Recommended Practice for Software Requirements Specification*, IEEE, 1998.
- [4] Liskov, B. *Data Abstraction and Hierarchy*, SIGPLAN Notices, 1988.
- [5] Haythorn, W. *What is Object-Oriented Design?*, Journal of Object-Oriented Programming, 1994, Vol. 7, nº 1.
- [6] Mednilla, N. and Gutierrez, I. *La incertidumbre como herramienta en la ingeniería del software*, Actas de las XI Jornadas de Ingeniería del Software y Bases de Datos. Sitges, España, 2006, 423-432.
- [7] Klir, G. J. and Folger, T. A. *Fuzzy Sets, Uncertainty and Information*, Prentice Hall, NJ, 1988.
- [8] *Merriam-Webster Online Dictionary*, <http://www.m-w.com/dictionary/abstract>.
- [9] Yourdon, E. and Constantine, L. L. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Prentice-Hall, 1979, 17-24.
- [10] Gamma, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995